Answer Sets and the Language of Answer Set Programming

Vladimir Lifschitz

■ *Answer set programming is a declarative programming paradigm based on the answer set semantics of logic programs. This introductory article provides the mathematical background for the discussion of answer set programming in other contributions to this special issue.*

Answer set programming (ASP) is a declarative programming paradigm introduced by Marek and Truszczyński (1999) and Niemelä (1999). It grew out of research on knowledge representation (van Harmelen, Lifschitz, and Porter 2008), nonmonotonic reasoning (Ginsberg and Smith 1988), and Prolog programming (Sterling and Shapiro 1986). Its main ideas are described in the article by Janhunen and Niemelä (2016) and in other contributions to this special issue.

In this introductory article my goal is to discuss the concept of an answer set, or stable model, which defines the semantics of ASP languages. The answer sets of a logic program are sets of atomic formulas without variables ("ground atoms"), and they were introduced in the course of research on the semantics of negation in Prolog. For this reason, I will start with examples illustrating the relationship between answer sets and Prolog and the relationship between answer set solvers and Prolog systems. Then I will review the mathematical definition of an answer set and discuss some extensions of the basic language of ASP.

Prolog and Negation as Failure

Simple Prolog rules can be understood as rules for generating new facts, expressed as ground atoms, from facts that are given or have been generated earlier. For example, the Prolog program

```
p(1). p(2). p(3).
q(2). q(3). q(4).
r(X) :- p(X), q(X).
```

consists of six facts ("1, 2, and 3 have property p; 2, 3, and 4 have property q") and a rule: for any value of X, r(X) can be generated if p(X) and q(X) are given or have been generated earlier.[1] In response to the query ?- r(X) a typical Prolog system will return two answers, first X = 2 and then X = 3.

Let us call this program Π_1 and consider its modification Π_2, in which the "negation as failure" symbol \+ is inserted in front of the second atom in the body of the rule:

```
p(1). p(2). p(3).
q(2). q(3). q(4).
r(X) :- p(X), \+ q(X).
```

The modified rule allows us, informally speaking, to generate r(X) if p(X) has been generated, assuming that any attempt to generate q(X) using the rules of the program would fail. Given the modified program and the query ?- r(X) Prolog returns one answer, X = 1.

What is the precise meaning of conditions of this kind, "any attempt to generate … using the rules of the program would fail"? This is not an easy question, because the condition is circular: it attempts to describe when a rule *R* "fires" (can be used to generate a new fact) in terms of the set of facts that can be generated using all rules of the program, including *R* itself. Even though this formulation is vague, it often allows us to decide when a rule with negation is supposed to fire. It is clear, for instance, that there is no way to use the rules of Π_2 to generate q(1), because this atom is not among the given facts and it does not match the head of any rule of Π_2. We conclude that the last rule of Π_2 can be used to generate r(1).

But there are cases when the circularity of the above description of negation as failure makes it confusing. Consider the following program Π_3, obtained from Π_2 by replacing the facts in the second line with a rule:

```
p(1). p(2). p(3).
q(3) :- \+ r(3).
r(X) :- p(X), \+ q(X).
```

The last rule justifies generating r(1) and r(2), there can be no disagreement about this. But what about r(3)? The answer is yes if any attempt to use the rules of the program to generate q(3) fails. In other words, the answer is yes if the second rule of the program does not fire. But does it? It depends on whether the last rule can be used to generate r(3) — the question that we started with.

The first precise semantics for negation as failure was proposed by Clark (1978), who defined the process of program completion — a syntactic transformation that turns Prolog programs into first-order theories. The definition of a stable model, or answer set, proposed ten years later (Gelfond and Lifschitz 1988), is an alternative explanation of the meaning of Prolog rules with negation. It grew out of the view that an answer set of a logic program describes a possible set of beliefs of an agent associated with this program; see the paper by Erdem, Gelfond, and Leone (2016) in this special issue. Logic programs are similar, in this sense, to autoepistemic theories (Moore 1985) and default theories (Reiter 1980).[2] The definition of an answer set, reproduced in this article, adapts the semantics of default logic to the syntax of Prolog.

We will see that program Π_3, unlike Π_1 and Π_2, has two answer sets. One answer set authorizes including X=3 as an answer to the query ?- q(X) but not as an answer to the query ?- r(X); according to the other answer set, it is the other way around. In this sense, program Π_3 does not give an unambiguous specification for query answering. Programs with several answer sets are "bad" Prolog programs.

In answer set programming, on the other hand, programs with several answer sets (or without answer sets) are quite usual and play an important role, like equations with several roots (or without roots) in algebra.

Answer Set Solvers

How does the functionality of answer set solvers compare with Prolog?

Each of the programs Π_1, Π_2, and Π_3 will be accepted as a valid input by an answer set solver, except that the symbol \+ for negation as failure should be written as not. Thus Π_2 becomes, in the language of answer set programming,

```
p(1). p(2). p(3).
q(2). q(3). q(4).
r(X) :- p(X), not q(X).
```

and Π_3 will be written as

```
p(1).  p(2).  p(3).
q(3) :- not r(3).
r(X) :- p(X), not q(X).
```

Unlike Prolog systems, an answer set solver does not require a query as part of the input. The only input it expects is a program, and it outputs the program's answer sets. For instance, given program Π_1, it will find the answer set

```
p(1) p(2) p(3) q(2) q(3) q(4) r(2) r(3)
```

From the perspective of Prolog, this is the list of all ground queries that would generate the answer yes for this program. For program Π_2, the answer set

```
p(1) p(2) p(3) q(2) q(3) q(4) r(1)
```

will be calculated. Given Π_3 as input, an answer set solver will find two answer sets:

```
Answer: 1
p(1) p(2) p(3) q(3) r(1) r(2)
Answer: 2
p(1) p(2) p(3) r(3) r(1) r(2)
```

Definition of an Answer Set: Positive Programs

I will review now the definition of an answer set, beginning with the case when the rules of the program do not contain negation, as in program Π_1 discussed earlier. By definition, such a program has a unique answer set, which is formed as follows.

First, we *ground* the program by substituting specific values for variables in its rules in all possible ways. The result will be a set of rules of the form

$$A_0 \text{ :- } A_1, \ldots, A_n. \tag{1}$$

where each A_i is a ground atom. (We think of "facts," such as p(1) in Π_1, as rules of form (1) with n = 0 and with the symbol :- dropped.) For instance, grounding turns Π_1 into

```
p(1). p(2). p(3).
q(2). q(3). q(4).
r(1) :- p(1), q(1).
r(2) :- p(2), q(2).
r(3) :- p(3), q(3).
r(4) :- p(4), q(4).
```

The answer set of the program is the smallest set *S* of ground atoms such that for every rule (1) obtained by grounding, if the atoms $A_1, \ldots, A_n$ belong to *S* then the head A_0 belongs to *S* too.

For instance, in the case of program Π_1 this set *S* includes (1) the facts in the first two lines of the grounded program, (2) the atom r(2), because both atoms in the body of the rule with the head r(2) belong to *S,* and (3) the atom r(3), because both atoms in the body of the rule with the head r(3) belong to *S*.

The following program contains two symbolic constants, block and table:

```
number(1). number(2). number(3).
location(block(N)) :- number(N).
location(table).
```

Grounding turns the second rule into

```
location(block(1)) :- number(1).
location(block(2)) :- number(2).
location(block(3)) :- number(3).
```

The answer set of this program consists of the atoms

```
number(1)  number(2)  number(3)  location(block(1))
location(block(2))  location(block(3))  location(table)
```

Definition of an Answer Set: Programs with Negation

In the general case, when the rules of the given program may contain negation, grounding gives a set of rules of the form

$$A_0 \text{ :- } A_1, \ldots, A_m, \text{ not } A_{m+1}, \ldots, \text{ not } A_n. \tag{2}$$

where each A_i is a ground atom. (To simplify notation, we showed all negated atoms at the end.) For instance, the result of grounding Π_2 is

```
p(1). p(2). p(3).
q(2). q(3). q(4).
r(1) :- p(1), not q(1).
r(2) :- p(2), not q(2).
r(3) :- p(3), not q(3).
r(4) :- p(4), not q(4).
```

To decide whether a set *S* of ground atoms is an answer set, we form the *reduct* of the grounded program with respect to *S,* as follows. For every rule (2) of the grounded program such that *S* does not contain any of the atoms $A_{m+1}, \ldots, A_n$, we drop the negated atoms from (2) and include the "positive part" (1) of the rule in the reduct. All other rules are dropped from the grounded program altogether. Since the reduct consists of rules of form (1), we already know how to calculate its answer set. If the answer set of the reduct coincides with the set *S* that we started with then we say *S* is an answer set of the given program.

For instance, to check that the set

$$\{p(1),\ p(2),\ p(3),\ q(2),\ q(3),\ q(4),\ r(1)\} \tag{3}$$

is an answer set of Π_2, we calculate the reduct of the grounded program with respect to this set. The reduct is

```
p(1). p(2). p(3).
q(2). q(3). q(4).
r(1) :- p(1).
```

(The last three rules of the grounded program are not included in the reduct because set (3) includes q(2), q(3), and q(4).) The answer set of the reduct is indeed the set (3) that we started with. If we repeat this computation for any set *S* of ground atoms other than (3) then the result may be a subset of *S,* or a superset of *S,* or it may partially overlap with *S,* but it will never coincide with *S*. Consequently (3) is the only answer set of Π_2.

Intuitively, the reduct of a program with respect to *S* consists of the rules of the program that "fire" assuming that *S* is exactly the set of atoms that can be generated using the rules of the program. If the answer set of the reduct happens to be exactly *S* then we conclude that *S* was a "good guess."

The concept of an answer set can be defined in many other, equivalent ways (Lifschitz 2010).

Extensions of the Basic Language

Arithmetic. Rules may contain symbols for arithmetic operations and comparisons, for instance:

```
p(1). p(2).
q(1). q(2).
r(X+Y) :- p(X), q(Y), X<Y.
```

The answer set of this program is

```
p(1) p(2) q(1) q(2) r(3)
```

(In view of the condition X < Y in the body, the only values substituted for the variables in the process of grounding are X = 1, Y = 2.)

Disjunctive Rules (Gelfond and Lifschitz 1991). The head of a rule may be a disjunction of several atoms (often separated by bars or semicolons), rather than a single atom. For instance, the rule

```
p(1) | p(2).
```

instructs the solver to include p(1) or p(2) in each answer set. The answer sets of this one-rule program are

```
Answer: 1
p(1)
Answer: 2
p(2)
```

Choice Rules (Niemelä and Simons 2000). Enclosing the list of atoms in the head in curly braces represents the "choice" construct: choose in all possible ways which atoms from the list will be included in the answer set. For instance, the one-rule program

```
{ p(1) ; p(2) }.
```

has 4 answer sets:

```
Answer:  1

Answer: 2
p(1)
 Answer: 3
p(2)
Answer: 4
p(1)   p(2)
```

A choice rule may specify bounds on the number of atoms that are included. The lower bound is shown to the left of the expression in braces, and the upper bound to the right. For instance, the one-rule program

```
1 { p(1) ; p(2) }.
```

has 3 answer sets — answers 2–4 from the previous example. The one-rule program

```
{ p(1) ; p(2) } 1.
```

has 3 answer sets as well — answers 1–3.

Constraints. A constraint is a disjunctive rule that has 0 disjuncts in the head, so that it starts with the symbol :-. Adding a constraint to a program eliminates the answer sets that satisfy the body of the constraint. For instance, the answer sets of the program

```
{ p(1) ; p(2) }.
:- p(1), not p(2).
```

are answers 1, 3, and 4 from the preceding list. Answer 2 violates the constraint, because it includes p(1) and does not include p(2).

Classical Negation (Gelfond and Lifschitz 1991). Atoms in programs and in answer sets can be preceded by the "classical negation" sign (–) that should be distinguished from the negation as failure symbol (not). This is useful for representing incomplete information. For instance, the answer set

```
p(a) p(b) -p(c) q(a) -q(c)
```

can be interpreted as follows: a and b have property p, and c does not; a has property q, and c does not; whether b has property q we do not know. A rule of the form

$– A$:- not A.

containing classical negation in the head and negation as failure in the body expresses the "closed world assumption" for the atom A: A is false if there is no evidence that A is true. The rule

```
p(T+1) :- p(T), not -p(T+1).
```

expresses the "frame default" (Reiter 1980) in the language of answer set programming: if p was true at time T and there is no evidence that p became false at time T + 1 then p was true at time T + 1.

Input languages of many answer set solvers include other useful extensions of the basic language, such as aggregates (Faber, Leone, and Pfeifer 2004), weak constraints (Buccafuri, Leone, and Rullo 1997), consistency-restoring rules (Balduccini and Gelfond 2003), and P-log rules (Chitta, Gelfond, and Rushton 2009).

Extending the Definition of an Answer Set

The problem of extending the definition of an answer set to additional constructs, such as those reviewed in the previous section, can be approached in several ways. One useful idea is to treat expressions in the bodies and heads of rules as logical formulas written in alternative notation. For instance, we can think of the list in the body of (2) as a conjunction of literals:

$$A_1 \wedge \cdots \wedge A_m \wedge \neg A_{m+1} \wedge \cdots \wedge \neg A_n.$$

A choice expression $\{A_1; \ldots ; A_n\}$ can be treated as a conjunction of "excluded middle" formulas:

$$(A_1 \vee \neg A_1) \wedge \cdots \wedge (A_n \vee \neg A_n)$$

(Ferraris and Lifschitz 2005). Under this approach, the rules of a grounded program are expressions of the form $F \leftarrow G$, where F and G are formulas built from ground atoms using conjunction, disjunction, and negation.[3]

The definition of the reduct was extended to such rules by Lifschitz, Tang, and Turner (1999). In the process of constructing the reduct of a rule $F \leftarrow G$ with respect to a set S of ground atoms, every subformula that begins with negation is replaced by a logical constant: by true if it is satisfied by S, and by false otherwise.

Gebser et al. (2015) defined the syntax and semantics of many constructs implemented in the solver CLINGO using a generalization of this approach that allows the formulas F and G to contain implication, and that allows conjunctions and disjunctions in F and G to be infinitely long.

Acknowledgements

Thanks to Gerhard Brewka, Martin Gebser, Michael Gelfond, Tomi Janhunen, Amelia Harrison, Amanda Lacy, Yuliya Lierler, Nicola Leone, and Mirek Truszczyński for comments on a draft of this article. This research was partially supported by the National Science Foundation under Grant IIS-1422455.

Notes

1. In Prolog programs, a period indicates the end of a rule. Capitalized identifiers are used as variables. The symbol :- reads "if"; it separates the "head" of the rule (in this case, the atom r(X)) from its "body" (the pair of atoms p(X), q(X)). Answer set programming inherited from Prolog these syntactic conventions and terminology.

2. The relationship between Prolog and autoepistemic logic was described by Gelfond (1987).

3. A more radical version of this view is to think of the whole rule $F \leftarrow G$ as a propositional formula — as the implication $G \rightarrow F$ "written backwards" (Ferraris 2005). It is also possible to avoid the reference to grounding in the definition of an answer set and to treat rules with variables as first-order formulas (Ferraris, Lee, and Lifschitz 2011).

References

Balduccini, M., and Gelfond, M. 2003. Logic Programs with Consistency-Restoring Rules. Paper presented at the 2003 AAAI Spring Symposium on Logical Formalizations of Commonsense Reasoning, 24–26 March, Stanford University, Stanford CA.

Buccafuri, F.; Leone, N.; and Rullo, P. 1997. Enhancing Disjunctive Datalog by Constraints. *IEEE Transactions on Knowledge and Data Engineering* 12(5): 845–860. dx.doi.org/10.1109/69.877512

Chitta, B.; Gelfond, M.; and Rushton, N. 2009. Probabilistic Reasoning with Answer Sets. *Theory and Practice of Logic Programming* 9(1): 57–144. dx.doi.org/10.1017/S1471068408003645

Clark, K. 1978. Negation as Failure. In *Logic and Data Bases,* ed. H. Gallaire and J. Minker. New York: Plenum Press. 293–322. dx.doi.org/10.1007/978-1-4684-3384-5_11

Erdem, E.; Gelfond, M.; and Leone, N. 2016. Applications of ASP. *AI Magazine* 37(3).

Faber, W.; Leone, N.; and Pfeifer, G. 2004. Recursive Aggregates in Disjunctive Logic Programs: Semantics and Complexity. In *Logics in Artificial Intelligence, 9th European Conference* (JELIA). Lecture Notes in Computer Science 3229. Berlin: Springer.

Ferraris, P. 2005. Answer Sets for Propositional Theories. In *Proceedings of Logic Programming and Nonmonotonic Reasoning, 8th International Conference* (LPNMR 2005). Lecture Notes in Computer Science 3662, 119–131. Berlin: Springer. dx.doi.org/10.1007/11546207_10

Ferraris, P., and Lifschitz, V. 2005. Weight Constraints as Nested Expressions. *Theory and Practice of Logic Programming* 5(1–2): 45–74. dx.doi.org/10.1017/S1471068403001923

Ferraris, P.; Lee, J.; and Lifschitz, V. 2011. Stable Models and Circumscription. *Artificial Intelligence* 175(1): 236–263. dx.doi.org/10.1016/j.artint.2010.04.011

Gebser, M.; Harrison, A.; Kaminski, R.; Lifschitz, V.; and Schaub, T. 2015. Abstract Gringo. *Theory and Practice of Logic Programming* 15(4–5): 449–463. dx.doi.org/10.1017/S1471068415000150

Gelfond, M. 1987. On Stratified Autoepistemic Theories. In *Proceedings of 6th National Conference on Artificial Intelligence* (AAAI), 207–211. San Mateo, CA: Morgan Kaufmann, Publishers.

Gelfond, M., and Lifschitz, V. 1988. The Stable Model Semantics for Logic Programming. In *Proceedings of International Logic Programming Conference and Symposium,* ed. R. Kowalski and K. Bowen, 1070–1080. Cambridge, MA: The MIT Press. dx.doi.org/10.1007/BF03037169

Gelfond, M., and Lifschitz, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing* 9(3–4): 365–385.

Ginsberg, M., and Smith, D. 1988. Reasoning About Action I: A Possible World Approach. *Artificial Intelligence* 35(3): 165–195. dx.doi.org/10.1016/0004-3702(88)90011-2

Janhunen, T., and Niemelä, I. 2016. The Answer Set Programming Paradigm. *AI Magazine* 37(3).

Lifschitz, V. 2010. Thirteen Definitions of a Stable model. In *Fields of Logic and Computation: Essays Dedicated to Yuri Gurevich on the Occasion of his 70th Birthday.* Lecture Notes in Computer Science Volume 6300, 488–503. Berlin: Springer. dx.doi.org/10.1007/978-3-642-15025-8_24

Lifschitz, V.; Tang, L. R.; and Turner, H. 1999. Nested Expressions in Logic Programs. *Annals of Mathematics and Artificial Intelligence* 25(3–4): 369–389. dx.doi.org/10.1023/A:1018978005636

Marek, V., and Truszczyński, M. 1999. Stable Models and an Alternative Logic Programming Paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective,* ed. K. Apt, V. W. Marek, M. Truszczyński, D. S. Warren. Berlin: Springer Verlag. 375–398. dx.doi.org/10.1007/978-3-642-60085-2_17

Moore, R. 1985. Semantical Considerations on Nonmonotonic Logic. *Artificial Intelligence* 25(1):75–94. dx.doi.org/10.1016/0004-3702(85)90042-6

Niemelä, I. 1999. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. *Annals of Mathematics and Artificial Intelligence* 25(3–4): 241–273. dx.doi.org/10.1023/A:1018930122475

Niemelä, I., and Simons, P. 2000. Extending the Smodels System with Cardinality and Weight Constraints. In *Logic-Based Artificial Intelligence,* ed. J. Minker. Dordrecht, The Netherlands: Kluwer. 491–521. dx.doi.org/10.1007/978-1-4615-1567-8_21

Reiter, R. 1980. A Logic for Default Reasoning. *Artificial Intelligence* 13(1–2): 81–132. dx.doi.org/10.1016/0004-3702(80)90014-4

Sterling, L., and Shapiro, E. 1986. The Art of Prolog: Advanced Programming Techniques. Cambridge, MA: The MIT Press.

van Harmelen, F.; Lifschitz, V.; and Porter, B., eds. 2008. *Handbook of Knowledge Representation.* Amsterdam: Elsevier.

Vladimir Lifschitz is a professor of computer science at the University of Texas at Austin. His research interests are in computational logic and knowledge representation.

James Crawford, Conference Chair
G. Michael Youngblood, Conference Cochair

The Twenty-Ninth Annual Conference on Innovative Applications of Artificial Intelligence

(IAAI-17)

February 4–9, 2017

San Francisco, California USA

Please Join Us!

www.aaai.org/iaai17